IATO: A Flexible EPIC Simulation Environment

Amaury Darsch
IRISA/INRIA
Campus de Beaulieu
35042 Rennes Cedex, France
adarsch@irisa.fr

André Seznec
IRISA/INRIA
Campus de Beaulieu
35042 Rennes Cedex, France
seznec@irisa.fr

Abstract

High-performance superscalar processors are designed with the help of complex simulation environment. The simulation infrastructure permits to validate the processor instruction set and contributes as well to the performance evaluation of the selected microarchitecture. Unfortunately, new architectures like the EPIC are not properly supported in the research community. Due to its specificity, the EPIC architecture requires a new framework that gives the researcher an opportunity to explore the EPIC paradigm by characterizing the static and dynamic behavior of binary programs. In particular, this task is made difficult by the fact that the EPIC architecture defines a fully predicated ISA.

This paper presents a novel simulation infrastructure, called IATO, that permits to analyze, emulate and simulate the EPIC microarchitecture by using the IA64 ISA as the reference architecture. The novelty of the environment is to provide an in-order and an out-of-order cycle accurate execution-driven simulators. In particular, the out-of-order simulator provides an innovative solution for the out-of-order execution of a fully predicated ISA.

1. Introduction

Modern superscalar processors are reaching the billion of transistors and operate at frequencies above the GHz. Their designs are the result of clever engineering techniques with one being the simulation methodology [13]. During the design process, a simulator is used to validate the processor microarchitecture as well as characterizing its performance. Because a simulation engine is cheap to develop and can be modified easily, the simulation technique has become an integral part of the microarchitecture research methodology [18].

A simulator can be either a functional simulator or a timing simulator which is driven either by traces or by execution[14, 19]. A functional simulator computes the true register values along the course of the simulation. A timing simulator evaluates the timing performances of one or several components of the processor or the whole system as well. A functional simulator is used to characterize the behavior of the processor while the timing simulator characterizes the temporal operating envelope.

With a trace-driven simulator, the simulator operates with a sequence of information that represent the instruction sequence. The instruction sequence is the image (i.e the trace) of the instructions being executed by the processor being simulated. With an execution-driven simulator, the simulator operates with an executable (i.e a binary program) and its input vary along with the execution of the executable. Some authors also use the term execution-driven simulation to refer to a simulation methodology that rely on the actual execution of some parts of the program. During the program execution, the sequence of interest is simulated with the rest of the program being executed naturally [22]. A trace-driven simulator is simple and can be easily developed and debugged. Although, a trace-driven simulator can easily model the processor behavior, it cannot reflect the internal operations of the processor. An execution-driven simulator is by far more accurate than a trace-driven simulator. An execution-driven simulator is also a functional simulator and provides key information about the processor's internal operations.

This paper presents a novel simulation environment called *IATO* that is designed to operate with the EPIC architecture by using the IA64 ISA. The novelty of the paper is to introduce an in-order simulator and an out-of-order simulator. These simulators are functional and execution-driven simulators that operate with IA64 binaries. *IATO* is freely available to the research community [7, 8].

This paper is organized as follows. Section 2 provides a background review of the currently available simulators. Section 3 presents the *IATO* environment and the methodology of use. Section 4 provides an additional description of the design and implementation of the in-order and out-

of-order simulator. Section 5 reports some experimental results and Section 6 concludes this paper.

2. Background

There exist numerous simulators that are either trace-driven or execution-driven[20]. Depending on the nature of the simulation, a particular simulation architecture might be more suitable than another one. If performance is the ultimate objective, a trace-driven simulator might be the appropriate answer. On the other hand, if accuracy is required, an execution-driven, or even, a timing-driven simulator might be required.

2.1. Trace-driven simulation

Trace-driven simulators are not functional simulators and do not incorporate timing stream. Trace-driven simulators operate with true results and are unable to model speculative paths. Trace-simulators are rather limited to specific component analysis such like cache or branch prediction simulators. Typical cache simulators are *CacheSim5* or *Dinero IV* [12]. *CacheSim5* is part of the *Shade* package [21] available from SUN Microsystems and *Dinero IV* is available from the University of Wisconsin.

2.2. Execution-driven simulation

The most notorious execution-driven simulator is the *SimpleScalar* tool suite [10]. *SimpleScalar* is a complex environment that provides an in-order and an out-of-order simulators. *SimpleScalar* is available with a variety of different ISA, including the Alpha ISA and is widely used in the research community. The *SimpleScalar* in-order simulator is a functional simulator augmented with a cache memory hierarchy and a state-of-the-art branch prediction system. The out-of-order simulator operates with the in-order simulator as a front-end and then uses the trace results to emulate the out-of-order back-end. *SimpleScalar* is written in C and can execute user-level program including system calls that are passed to the host machine.

There exist simulators for the most popular ISAs. *ABSS* is a SPARC based multi-processor simulators [11]. *PSim* is a simulator that implements the PowerPC ISA [4]. *Psim* was extended with a performance simulator called *MW* in order to build a tightly coupled functional performance simulator called *fMW* [9]. *RSIM* [23], the Rice simulator, implements the MIPS R1000 ISA. All of these simulators are written either in C or C++ and most of them are available in the public domain.

Another class of simulator is the *system simulation* class that permits to model the operation of a complete system. Trace-driven or execution-driven simulators generally simulate only the processor and the memory hierarchy without any consideration of the surrounding environment. However, it is sometimes required to incorporate during the course of the simulation the I/O effects as well as the operating system activity. The *SimOS* system [16, 15] is the typical example of such system simulator. Designed for the MIPS architecture, *SimOS* can boot an operating system and can be configured with various processor models ranging from a simple pipelined model to an accurate superscalar microarchitecture. Another system simulation is the *SIMICS* [17] designed for the SPARC and Alpha architecture or the *Bochs* [5] emulation environment designed for the X86 processor.

2.3. Overview of the IA64 ISA

The INTEL/HP EPIC IA64 ISA[2, 3] is a complex ISA that features large register files and predicated instructions. Each instruction is associated with a type that categorizes the instruction function and the execution unit. The execution units are the *Memory* (M), the *Integer* (I), the *Floating-point* (F) and the *Branch* (B). The instructions are grouped into a 128-bits container called a *bundle*. There are generally 3 instructions per bundle. Almost all instructions are predicated with zero, one or two destinations. The IA64 ISA defines several large register files that comprises 128 *general registers* (GR), 128 *floating-point registers* (FR), 64 *predicate registers* (PR), 128 *application registers* (AR) and other book-keeping registers. The IA64 ISA defines two modes of operations for certain classes of registers. Some are said to be *stacked* (i.e window based) and other are said to be *rotating* (within a pipelined loop). The GR are stacked and rotating registers (from 32 to 127). The FR are rotating registers (from 32 to 127) as well as the PR (from 16 to 63). Both stacked and rotating registers are under the control of the *register stack engine* (RSE) whose state is contained in the *current frame marker* (CFM) register. Other registers are mostly control registers. During the execution, stacked and rotating registers are logically renamed by the RSE. The IA64 ISA also defines special registers that are built by combining several ISA registers. For instance, the *all predicate register* is a 64 bits register that is built by coalescing all 64 predicate registers.

2.4. Infrastructure challenges

The design of an EPIC in-order simulator poses some technical problems as the pipeline organization and execution is driven by a whole stage line. Furthermore, the design of an out-of-order simulator requires innovative solutions as the problem of executing predicated instructions with an out-of-order core is still an open problem.

3. The IATO environment

IATO [8, 7] (IA64 Toolkit) is a research environment that permits to analyze, emulate or simulate the IA64 Instruction Set Architecture (ISA) binary executables. *IATO* is a flexible and portable toolkit that is built around a set of C++ libraries and client programs. The *IATO* clients are applications programs based on the *IATO* libraries. The fundamental clients are the IA64 emulator and simulators. Other clients provide support for program analysis and statistical computation. Table 1 shows the applications clients available in the *IATO* distribution.

Client	Description
IAOS	An IA64 program analyzer and dumper.
IAKA	An IA64 functional emulator.
IAIO	An IA64 in-order simulator.
IAOO	An IA64 out-of-order simulator.
IATA	A trace and statistic analyzer.

Table 1. The IATO clients.

The *IATO* environment is supported on various platforms, including Linux based IA64 machines. Linux IA32 and Sun SPARC Solaris host machines are supported as well. The *IATO* environment is designed to operate with binary executable compiled for an UNIX environment. Popular IA64 compilers like GCC (Gnu C compiler) or ECC (Intel Electron compiler) are supported at the execution level.

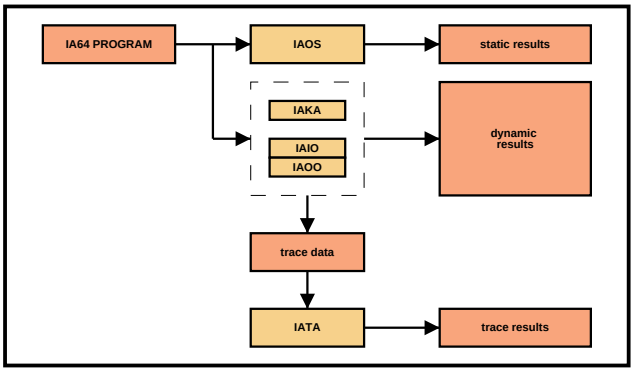

Figure 1. IATO operating flow.

Figure 1 illustrates the *IATO* operating flow. With an IA64 program object, the *IAOS* client produces static results. The *IAKA* emulator and the *IAIO, IAOO* simulators produce dynamic analysis or execution traces. The execution trace can be easily processed by the *IATA* client to produce new traces or other execution results.

3.1. IAOS client program

IAOS is a simple client program that produces static information from a binary executable. The resulting information comprises, among other things, the disassembled object code, the static instruction count and the predication rate. An extensive instruction distribution analysis can also be performed. The primary reason to use the *IAOS* client is to collect information about an IA64 binary executable.

3.2. IAKA client program

IAKA is a client program that emulates and traces IA64 binary executables. The emulator executes the common loop: fetch, decode and execute. During the execution, the emulator computes the true results for each instruction. System calls are emulated by transferring the control to the host machine. After each bundle execution, a trace result can be produced with the instructions, the memory and register accesses.

3.3. IAIO client program

IAIO is a client program that simulates and traces binary IA64 executables. The *IAIO* client is an execution-driven simulator that implements an 8-stages in-order pipeline. The pipeline organization is similar to the Itanium 2 microarchitecture [3]. During the course of the simulation, a stage by stage trace can be generated as well as a summarized statistic. The simulator is primarily used to computed the program IPC.

3.4. IAOO client program

IAOO is a client program that simulates and traces binary IA64 executables. The *IAOO* simulator implements a 10 stages out-of-order core engine, with true predicated instruction execution based on *Translation Register Buffer* (TRB) [6]. The simulator is cycle accurate and provides numerous architectural options that permits to perform a resource fine tuning. Like the *IAIO* client, the simulator can produce traces and statistics like the program IPC.

3.5. IATA client program

The *IATA* client program analyzes and prints binary traces produced by the other clients. It can also generate one trace from another. Trace generation can be seen as a post-processing operation. Another useful feature is the statistic computation. Given a trace file, the *IATA* client can extract

for example, the instruction distribution or other runtime information including the program IPC.

3.6. Program object information

The *IAOS* client is the essential client that permits to collect static information from an IA64 binary executable. The exact format of the IA64 binary executable is discussed in Section 3.8.

Disassembly: the code object disassembly is the primary function performed by the *IAOS* client. The resulting information can be selectively formatted, but usually comes in the form shown below. Example 1 shows a program dump obtained with the SPEC 2000 *gzip* program (cf. Section 5) compiled with the Intel Electron Compiler (ECC).

```
0x4000000000000660 (p01) st2 [r40]=r9
                         mov.i ar.pfs=r34;;
                         mov b0=r35;;
```

Example 1. Program object disassembly.

Instruction statistics: the static analysis for an IA64 binary permits to get a representation of the program contents with an emphasis on the predicate distribution. The difference between the number of instructions and the number of *nop* instructions represents the number of useful instructions. Also important is the number of non-branch predicated instructions with respect to the number of predicated instructions.

```
number of instructions         : 138885
number of usefull instructions : 99142
nop instructions               : 39743
predicated instructions        : 19348
predicated instructions non br : 10838
```

Example 2. Program instruction count.

Instruction nop: the instruction `nop` distribution is another important parameter with the EPIC architecture. Example 3 illustrates the static instruction nop distribution for each unit with the *gzip* SPEC program.

3.7. Trace production and analysis

The *IAKA* emulator and the *IAIO*, *IAOO* simulators can produce a result *trace file*. Since true results are computed, the trace includes the executed instructions, the register read and write operations, the memory access as well as the canceled instructions. With an emulator, the trace is produced

```
number of nop instructions [M] : 7574
number of nop instructions [I] : 25758
number of nop instructions [F] : 3574
number of nop instructions [B] : 2837
number of nop instructions     : 39743
```

Example 3. Program object nop distribution.

by one source which is the emulator itself. With a simulator, the trace can be produced by any pipeline stages. However, most of the time, the trace comes from the *commit* stage.

```
trace 408 {
EMU:RINSTR @0x4000000000003c60 mov r46=r37
EMU:RINSTR @0x4000000000003c60 ld8 r14=[r15]
EMU:RINSTR @0x4000000000003c60 nop.b 0x0;;
}
```

Example 4. Simulation trace dump.

The traces are stored in a binary file that can become huge when doing large simulations. The trace file can be further processed by the *IATA* client to produce dynamic runtime statistics. Traces can also be used to analyze special operations such like register read and write or memory access. The example above shows a trace record produced by the *IATA* client. The trace identification is built with two names. The first one is the record source name (EMU), and the second one is the record type (RINSTR). The record source name is the simulation object that produces the record. For example, `EMU` is a record produced by the emulator. The record type is an identification of the record. For example, `RINSTR` is a *raw instruction*. Another important feature of the *IATA* client program is the trace transformation. Given a program trace, under certain condition, another trace can be generated. This kind of post-processing permits to model the impact of certain resources by rescheduling the instruction stream. Although this approach is not as accurate as an execution-driven simulator, it is a useful feature that operates quickly.

3.8. Binary execution model

The *IATO* environment operates with full IA64 ELF binary that conforms to the Intel ABI specification[1]. As of today, the Linux environment supports such configuration for the UNIX environment. For a practical point of view, the *IATO* environment operates with statically linked binaries. In fact, there is really no restriction to operate with dynamically linked binary executables, except that it would complicate the overall system execution and add some over-

head that is not really needed for the analysis. The execution model is designed to be identical with a real one. For this reason, the process image that is built in memory and the memory that is emulated are almost identical to the one found in a real machine. Figure 2 shows the memory image that is built by the emulator or the simulators.

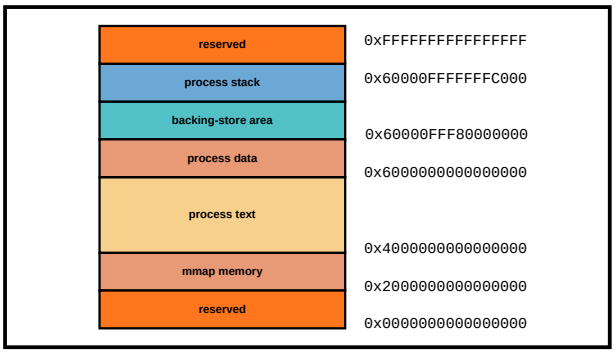

Figure 2. IA64 simulation memory model.

When the program is started, the program's arguments are passed by building a stack representation as it would appear on a real system. The stack is initialized with the *auxiliary vector table*, the *environment variables* and the *program arguments*. Opposite to the stack (that grows downward), the memory *backing store area* (BSA) is created and the processor registers initialized with the right values. Note that the BSA is used by the RSE during the spilling and filling operations. The BSA grows upward. The memory allocation is performed by the program with the help of two system calls that are emulated by the *IATO* environment. The first system call is `sbrk` which is used to control the process break limit. The second system call is `mmap` that is used to allocate memory pages. Note that the traditional memory allocation function `malloc` can use one call or the other, depending on the memory request size.

4. Simulator architecture

The *IATO* environment provides three different methods to execute an IA64 binary program. The *IAKA* client is the emulator that acts as a functional simulator. The *IAIO* client is the in-order simulator and the *IAOO* client is the out-of-order simulator.

4.1. IAKA functional simulation

The *IAKA* client is the functional simulator that permits to quickly execute an IA64 binary program. The emulator executes the common loop *fetch, decode, execute* as shown with Figure 3 but do not incorporate timing information and do not take into account resource constraints. The emulator is primarily used for trace production. It can also be used to validate committed information produced by the other simulators. The emulator supports the program system calls by executing them on the host machine.

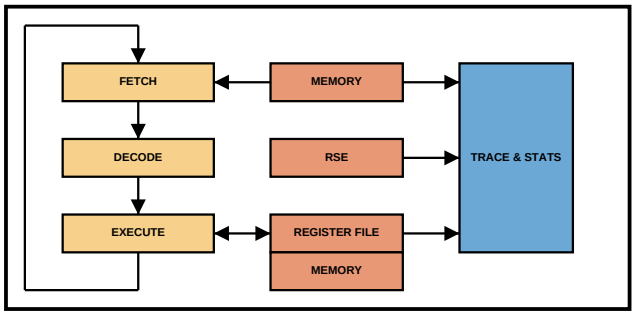

Figure 3. Emulator main loop.

Since the emulator is a functional simulator, true results are computed by the execution engine. This functional execution engine is common to all simulators and operates with the machine representation. This is particularly true for the floating-point numbers those special representation in the IA64 requires special adaptation. As a consequence, the computed results are similar to the one computed on the real machine, but this accuracy reduces a little the simulation performances. The emulator can also be used to quickly analyze the behavior of specific resources like branch predictors. The interesting part is that a simple fetch mechanism that mimics the one found in an in-order simulator can be implemented in the emulator. By adding cycle penalty during simulated branch misprediction, an extremely fast simulation trace can be built. This method is not as accurate as the in-order simulator since the method does not take into account the floating-point latency, it is particularly useful to get quickly the best in-order program IPC achievable by an EPIC machine.

4.2. IAIO, the in-order simulator

The *IAIO* client is the EPIC in-order simulator. Figure 4 illustrates the implementation of the simulator that is similar to the Itanium 2 microarchitecture.

The simulator is built with an 8-stages pipeline. The pipeline is split between a front-end part and a back-end part. The front-end part fetches the instruction bundles and rotate them before their expansion. The front-end and back-end are separated by a decoupling buffer that can hold 8 instruction bundles. The front-end part is composed of the *Instruction Pointer Generation* (IPG) and *Rotate* (ROT)

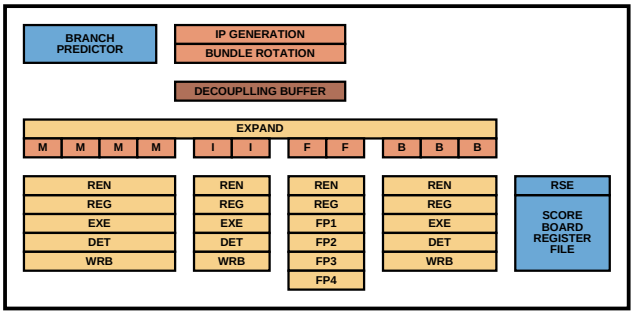

Figure 4. In-order simulator pipeline.

stages. Once the instruction has been rotated, the decoupling buffer stores the instructions bundles prior their expansion in the *expand* (EXP) stage. The expansion stage places the instruction into their respective slot. For the Itanium 2, there are 11 slots (resp. 4M, 2I, 2F and 3B slots). The simulator provides several options that controls the number of units, the resources size and eventually the unit latency.

Conceptually, the simulator is implemented as a double-linked list of stages. The back-end part uses micro-pipeline for each slot that are synchronized horizontally. If one stage is blocking the whole pipeline line is blocking as well. The double-linked list permits to simulate a fully-interlocked pipeline. The decoupling buffer acts as an interface between the front-end and the back-end. If the back-end is holding, the front-end continues to fetch instructions as long as the decoupling buffer is not full. An one cycle simulation is done in reverse order, starting from the last stage up-to the first one. This technique is similar to the one found in [14] but special care must be taken for the horizontal line synchronization. The resource synchronization is primarily controlled by the register scoreboard. Since each instruction is dispatched into its micro-pipeline during the expand stage, the scoreboard organization is considerably simplified. Actually, the scoreboard engine controls only the register availability in the *register read* (REG) stage. If one register is not available, the stage is put on hold, and as a consequence, the whole pipeline line is blocked as well. The back-to-back instruction execution is done with two bypass networks called the *early* and *late* bypass networks. The early bypass operates within the *execute* (EXE) stage while the *late* bypass operates within the *detect* (DET) stage. The *detect* (DET) stage is by far one of the most complex stage. The stage is the last stage for processing the exceptions as well as detecting the mis-speculation conditions. After the *detect* stage, all instructions which are not canceled are guaranteed to commit their value. For this reason, it is also the *detect* stage that computes the pipeline flush conditions. Note that a pipeline flush removes the instructions after the detection point and all instructions in the previous stages.

4.3. IAOO, the out-of-order simulator

The *IAOO* client is an innovative out-of-order simulator. The simulator implements a 10-stages out-of-order core engine, with true predicated instruction execution based on an innovative predicate execution system. The simulator is a cycle-accurate, execution-driven engine that features the same resources available in the Itanium 2 original implementation. In particular, the number of units is identical.

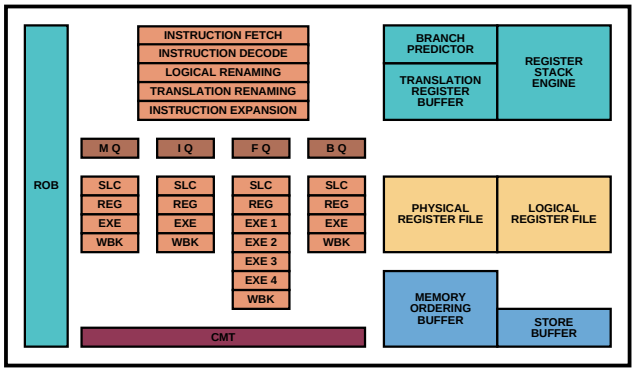

Figure 5. Out-of-order simulator pipeline.

Figure 5 illustrates the simulator main pipeline. The out-of-order simulation model mimics a classical out-of-order superscalar processor. Some logic pieces have been adapted to fit the IA64 requirements as well as the needs to handle properly the effects induced by the use of predicates. A complete description about the predicate engine (i.e TRB based engine) can be found in [6]. The simulation model implements four pipeline types, each corresponding to the IA64 instruction type (M, I, F and B). Associated with the pipeline is a set of resources that interacts with the 4 pipelines. Except for the TRB part, most of these resources are found in classical out-of-order microarchitecture. The first noticeable difference is the presence of two renaming stages. The REN stage is the logical renaming stage while the TRA stage is the translation renaming stage. The second difference with usual out-of-order pipeline, is the absence of a specific memory read stage since the IA64 ISA does not combine an address computation and a memory access in a single instruction. The simulator microarchitecture operates with a dual register file. A *Physical Register File* (PRF) is used to store the in-flight instruction results while a *Logical Register File* (LRF) is used to store the committed instruction results.

The out-of-order simulator implementation is similar to the previous one, used for the in-order simulator. Note that

Stage	Description
IF	Instruction fetch
DEC	Decode
REN	Logical renaming
TRA	translation renaming
EXP	Instruction expansion
SLC	Instruction selection
REG	Register read
EXE	Instruction execution
WBK	Register write-back
CMT	Instruction commit

Table 2. Pipeline stages summary.

it is the simulator implementation and not the microarchitecture which is similar. The microarchitecture uses micropipelines for each unit as described before, but the comparison stops here. After the instructions have been expanded into their respective queue, the instructions are placed into a reservation station table and wait for their operands to be available. The reservation station tables replace the scoreboard which is used in the in-order simulator. There is one reservation table for each unit. The instruction selection is performed when all operands are ready without considering the instruction predicate. The predicate availability is checked in the *register read* (REG) for all instructions except the instruction in the B unit which are checked in the *execute* (EXE) stage. If the predicate is not ready, it can either be predicted or the instruction is rescheduled. Instruction rescheduling is an essential part of the out-of-order microarchitecture. Unlike the in-order simulator, a global bypass network is used to perform the back-to-back operations. The physical register number (i.e the TRB number) is used as a tag during the data broadcast. The bypass network is updated if the instruction is not canceled. During the broadcast operation, the reservation stations update their ready flag. In some cases, it is necessary to wake-up instructions which have operands that are never written by canceled instructions. The B unit instructions check the predicate in the *execute* (EXE) stage because the IA64 ISA permits to forward, in the same cycle, a predicate from another unit to the branch unit. It is the bypass network which is used to implement such forwarding. The *write-back* (WRB) stage, while updating the physical register file, also checks the speculative conditions associated with the instruction. If the instruction is in a bad speculative state, the pipeline will be flushed in the next cycle. Because, there is a separation between the logical and the physical register files, there is no danger to update the physical register file in parallel of the speculative verification. It is also during the *write-back* stage that the *reorder buffer* is updated in order to mark the instruction as almost completed. In some respect, the *write-back* stage is similar to the *Detect* (DET) stage of the in-order microarchitecture.

5. Experimental results

In this section, experimental results are presented. These results were obtained with a subset of the SPEC 2000 Integer benchmarks. All benchmarks were compiled with the Intel Electron Compiler (ECC), version 7.1 using the O3 optimization level.

5.1. Static analysis

The static analysis is performed with the *IAOS* client. Table 3 shows the static analysis for the selected benchmarks, with the number on instructions (instr), the percentage of predicated instruction (pred) and the percentage of predicated instructions that are not branches (nbrp). All percentage are absolute. The static analysis is particularly important as it illustrates the compiler behavior. It permits also to get a quick image of the ratio between the branch and non-branch instructions that are predicated.

name	instr	pred	nbrp
gzip	138885	13.9%	7.80%
bzip2	135743	14.0%	7.85%
mcf	127122	14.2%	7.90%
parser	164474	12.9%	6.75%
twolf	244618	13.1%	8.39%
crafty	194277	12.8%	7.13%
vpr	196699	12.0%	6.96%

Table 3. SPEC 2000 integer static analysis.

5.2. Dynamic analysis

The dynamic analysis is performed with the *IAKA* client. *IAKA* is used here to gather the instruction execution as well as the predicate behavior. Table 4 shows the bundle and instruction count along with the *nop* rate for each of the selected benchmark. The instruction count ranges from 200M to 3B instructions with an average *nop* rate of 20%. Table 5 show also the predicate statistics for each of the selected benchmarks. The predication rate (pred), the non-branch instruction rate (nbrp), the cancellation rate (canc) and the non-branch cancellation rate (nbcn) is reported. Except for *bzip2*, the predication rate is around 15% on average with 9% corresponding to non-branch predicated instructions. This suggests that predicates are equally distributed between branch and non-branch instructions.

name	bundles	instr	nop
gzip	93641901	239295983	24.5%
bzip2	347962056	775463894	21.0%
mcf	117094708	346181078	27.3%
parser	190561186	568360378	24.5%
twolf	190597558	559177273	29.4%
crafty	270026048	807255596	18.5%
vpr	979042612	2894438352	24.9%

Table 4. SPEC 2000 integer dynamic analysis.

name	pred	nbrp	canc	nbcn
gzip	24.1%	18.3%	10.5%	7.46%
bzip2	44.5%	42.5%	2.63%	1.45%
mcf	15.5%	8.3%	5.42%	2.75%
parser	16.4%	9.4%	8.27%	3.59%
twolf	14.9%	11.7%	6.15%	4.45%
crafty	12.3%	6.3%	6.45%	2.86%
vpr	16.8%	14.0%	9.99%	8.35%

Table 5. SPEC 2000 dynamic analysis.

6. Conclusion

A complete EPIC/IA64 simulation environment was presented in this paper. The environment is freely available to the research community with several client applications. Although the development of simulation environment is a well-known technique, the specificity of the EPIC/IA64 ISA make this task a challenging issue, especially with the out-of-order simulation engine. To do so, a novel predicated instruction execution engine has been successfully implemented and validated. In the future, the simulator will be enhanced to support a predicate prediction mechanism in order to support research work in that direction. This work is planned to be coupled with a selective predicate prediction scheme.

References

[1] *Intel Itanium Architecture. Application Binary Interface*, 2000.

[2] *Intel Itanium Architecture. Software Developer's Manual*, 2000.

[3] *Intel Itanium Architecture. Itanium II microarchitecture*, 2002.

[4] PSIM PowerPC architectural simulator, http://ecos.sourceware.org/ecos/boards/psim.html, 2003.

[5] BOCHS: The cross-platform IA32 emulator, version 2.1, http://bochs.sourceforge.net, 2004.

[6] Amaury Darsch and André Seznec. Out-of-order Predicated Execution with Translation Register Buffer. *Internal report 1573, IRISA*, 2003.

[7] Amaury Darsch, André Seznec and Pierre Villalon. *IATO, The IA64 Toolkit. Application programming interface, http://www.irisa.fr/caps/projects/ArchiCompil/iato*, 2004.

[8] Amaury Darsch, André Seznec and Pierre Villalon. *IATO, The IA64 Toolkit. Reference manual, http://www.irisa.fr/caps/projects/ArchiCompil/iato*, 2004.

[9] Candice Bechem, Jonathan Combs, Noppanunt Utamaphetai, Bryan Black, R.D. Shawn Blanton and John Paul Shen. An Integrated Functional Performance Simulator. *IEEE Micro*, 19(3):26–35, May 1999.

[10] Doug Burger and Tod M. Austin. *The SimpleScalar Tool Set, version 2.0*, June 1997.

[11] Dwight Sunada, David Glasco and Michael Flynn. ABSS v2.0: a SPARC simulator. In *Workshop on Synthesis And System Integration of Mixed Technologies*, 1998.

[12] J. Edler. Dinero IV Trace-Driven Uniprocessor Cache Simulator, http://www.cs.wisc.edu/ markhill/DineroIV.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture; A quantitative approach - Third edition*. Morgan Kaufman Publishers, 2003.

[14] Mayan Mougdil. Techniques for implementing fast processor simulators. In *Annual simulation symposium*, pages 83–90, Apr. 1998.

[15] Mendel Rosenblum, Edouard Bugnion, Scott Devine and Stephen Alan Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. *Modeling and Computer Simulation*, 7(1):78–103, 1997.

[16] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel and Anoop Gupta. Complete Computer System Simulation: The SimOS Approach. *"IEEE parallel and distributed technology*, 3(4):34–43, 1995.

[17] Peter Magnusson and Bengt Werner. Effi cient Memory Simulation in SimICS. In *Annual Simulation Symposium, Phoenix*, Apr. 1995.

[18] Pradip Bose. Performance evaluation and validation of microprocessors. *Sigmetrics*, pages 226–227, May 1999.

[19] Pradip Bose and Thomas M. Conte. Performance analysis and its impact on design. *IEEE Computer*, pages 41–49, May 1998.

[20] Renato J. Figueiredo, Jos A.B. Fortes, Rudolf Eigenmann, Nirav Kapadia, Valerie Taylor, Alok Choudhary, Luis Vidal and Jan-Jo Chen. On the Use of Simulation and Parallelization Tools in Computer Architecture and Programming Courses, 2000.

[21] Robert Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profi ling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137, May 1994.

[22] Thierry Lafage and André Seznec. Combining Light Static Code Annotation and Instruction-Set Emulation for Flexible and Effi cient On-the-Fly Simulation. In *International Euro-Par Conference, Munich*, pages 178–182, Sept. 2000.

[23] Vijay S. Pai, Parthasarathy and Sarita V Adve. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In *Workshop on Computer Architecture Education*, 1997.